DNA features viewer

Application Note

DNA Features Viewer, a sequence annotation formatting and plotting library for Python

Valentin Zulkower [1,*], Susan Rosser[1]

[1] Edinburgh Genome Foundry, SynthSys, School of Biological Sciences, University of Edinburgh, EH9 3BF Edinburgh, UK

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: While the Python programming language counts many Bioinformatics and Computational Biology libraries, none offers customizable sequence annotation visualizations with layout optimization.
Results: DNA Features Viewer is a sequence annotation plotting library which optimizes plot readability while letting users tailor other visual aspects (colors, labels, highlights, etc.) to their particular use case.
Availability: Open-source code and documentation are available on Github under the MIT licence (https://github.com/Edinburgh-Genome-Foundry/DnaFeaturesViewer).
Contact: valentin.zulkower@ed.ac.uk
Supplementary information: attached.

1 Introduction

DNA sequence visualization is a common need in Bioinformatics, and many software tools can plot sequence annotations from Genbank or General Feature Format (GFF) records. A sequence annotation specifies a location (start position, end position and strand), feature type (such as "CDS" or "regulatory") and attributes (e.g. gene name, species of origin, or locus tag). When displaying a record with many annotations, one may want to enhance readability by hiding or highlighting certain features and attributes to focus the reader's attention on the most relevant information.

Interactive sequence editing software such as SnapGene Viewer (www.snapgene.com) or Benchling (www.benchling.com) enable users to manually color or hide sequence features, but the customization is limited and cannot be automated. The `gff2ps` command-line interface (Abril and Guigo, 2000) converts GFF files to plots in PostScript format, and allows some level of automation and customization. Python modules for sequence plotting are scarce and lack automation capabilities, making them difficult to integrate with other projects (see Supplementary Section A for frameworks reviews and comparison). For instance, both DnaPlotLib (Der *et al.*, 2017) and Biopython (Cock *et al.*, 2009) require users to style each annotation separately, and do not automatically avoid collisions between overlapping annotations and their labels.

Here we present DNA Features Viewer, a Python library which lets users define visual "themes" determining the label and display style of each annotation as a function of its type, location, and attributes. Annotations are then automatically laid out to create compact and readable plots, making the library a robust choice as a generic plotter for other frameworks. Plots can be exported in PNG, SVG, PDF or interactive HTML format, for use in interactive notebooks, PDF reports, or web applications.

2 Usage and examples

2.1 Definition of visual themes

In DNA Features Viewer, sequence annotation records read from Genbank or GFF files are converted to so-called *graphic records*, which define the visual aspects of each annotation. The conversion is ensured by a user-defined Python class (the *translator*) whose attributes and methods indicate which annotations should appear in the plot (and which should be discarded), as well as the visual style of each annotation, including arrow color, arrow width, edge width, label text, associated label in the figure's legend, and text font properties. For instance, the translator class used in Figure 1A sets the label text as either the `\note` or `\gene` attribute of the annotation, assigns each feature's color based on the feature's type, and reports the color/type correspondence in the figure legend. A translator thus acts as a visual theme which can be defined once and used throughout a project to ensure style consistency across annotation plots.

2.2 Plot readability optimizations

Figure 1A also illustrates how DNA Feature Viewer automatically lays out the visual elements of a graphic record to optimize compactness and readability. Feature labels such as "backbone" and "GFP" are displayed directly inside their corresponding feature arrow, and the font color is

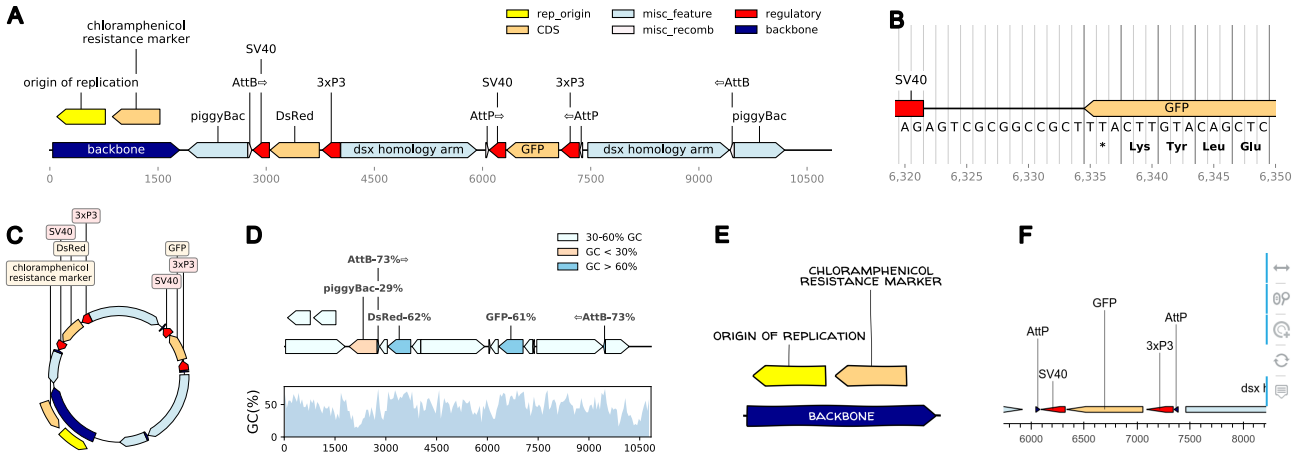


Fig. 1. Different views of a pBac cloning vector (Kyrou et al., 2018) plotted using DNA Features viewer. The Python code to generate each figure is provided in Supplementary Section C. (A) Plasmid map plot using a custom visual theme as described in Section 2.1. (B) Detail plot focusing on a short sequence segment, with nucleotide and amino-acid sequences, and vertical visual guides. (C) Circular view of the plasmid. In this visual theme, label text boxes are automatically colored to be easily associated with their corresponding features. (D) Plasmid plot with colors indicating the GC content at each feature's location. High- and low-GC features are highlighted with a label indicating their average GC content. The bottom subplot, which shares the same x-axis, indicates the local GC content over 100-nucleotides windows. (E) Plot using Matplotlib's path.sketch filter and a custom font to create a "handwriting" effect. (F) Interactive HTML plot generated via the Bokeh library (shown here with a zoom around the position at location 7000). Icons on the left refer to widgets enabling mouse-based interactions.

automatically selected (as black or white) to fit the feature's background color. Labels which do not fit inside a feature arrow are displayed above it, and wrapped on several lines when necessary (e.g. "chloramphenicol resistance marker"). For narrow features whose orientation cannot be easily discerned (such as AttB and AttP sites in Figure 1A), a text arrow is added to the label. Finally, all features and label texts are organized along different vertical levels to avoid collisions (the layout optimization method, which uses variant of graph coloring algorithm, is described in Supplementary Section B). This ensures that the resulting plot remains readable irrespective of the figure's width, which is set by the user and often constrained by space limitations on a web page or PDF report.

2.3 Other visualization formats

DNA Features Viewer supports a variety of plotting formats to suit different use cases. For instance, it enables to focus on on a small sequence region, displaying the nucleotide and amino-acid sequences (Figure 1B), or to plot the record's full sequence over multi-line, multipage PDF documents (as shown in Supplementary Section D). A record can also be displayed with a circular topology, with text labels on the top (Figure 1C).

The library relies primarily on the Matplotlib plotting framework (Hunter, 2007) for graphics rendering, making it possible to display sequence annotations along with other other data visualization. For instance DNA Features Viewer has been used to associate sequence maps with local ChIP RZ scores in Kroner *et al.* (2019), and local GC content in Greig *et al.* (2018) (also illustrated in Figure 1D). Matplotlib also allows to finely tune plotting style with custom fonts and path filters, as illustrated in Figure 1E, to suit different media (articles, presentation slides, etc.)

Finally, the Bokeh library (Bokeh Development Team, 2019) can be used as a plotting backend, although this support is limited to linear sequence views. This allows the rendition of graphic records as interactive HTML plots which can be integrated in a webpage and allow the exploration of very large features record thanks to interactive widgets to pan and zoom around local regions (as shown in Figure 1F).

3 Implementation

DNA Features Viewer is written in Python. Genbank file parsing is provided by the Biopython library, and GFF parsing by the BCBB library (https://github.com/chapmanb/bcbb, unpublished).

Funding

The Edinburgh Genome Foundry is supported by the BBSRC (BB/M025659/1, BB/M025640/1, and BB/M00029X/1 to SR) and the BBSRC/MRC/EPSRC funded UK Centre for Mammalian Synthetic Biology (BB/M0101804/1 to SR) as part of the RCUK's Synthetic Biology for Growth programme.

Acknowledgments:

We thank Yu-jin Kim for comments and suggestions.

References

Abril, J. F. and Guigo, R. (2000). Gff2ps: Visualizing genomic annotations. *Bioinformatics*.

Bokeh Development Team (2019). *Bokeh: Python library for interactive visualization*.

Cock, P. J. A. *et al.* (2009). Biopython: Freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics*, **25**(11), 1422–1423.

Der, B. S. *et al.* (2017). DNAplotlib: Programmable Visualization of Genetic Designs and Associated Data. *ACS Synthetic Biology*, **6**(7), 1115–1119.

Greig, D. R. *et al.* (2018). MinION nanopore sequencing identifies the position and structure of bacterial antibiotic resistance determinants in a multidrug-resistant strain of enteroaggregative Escherichia coli. *Microbial Genomics*, **4**(10).

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, **9**(3), 90–95.

Kroner, G. M. *et al.* (2019). Escherichia coli Lrp regulates one-third of the genome via direct, cooperative, and indirect routes. *Journal of Bacteriology*, **201**(3).

Kyrou, K. *et al.* (2018). A CRISPR-Cas9 gene drive targeting doublesex causes complete population suppression in caged Anopheles gambiae mosquitoes. *Nature biotechnology*, **36**(11), 1062–1066.

Supplementary Information to

DNA Features Viewer: an sequence annotation formatting and plotting library for Python

Valentin Zulkower [1, *], Susan Rosser [1]

[1] Edinburgh Genome Foundry, SynthSys centre for Synthetic and Systems Biology, School of Biological Sciences, University of Edinburgh, EH93BF Edinburgh

[*] valentin.zulkower@ed.ac.uk

Content of the Supplementary Information

A. Other annotation plotting frameworks

In this section we compare different Python sequence annotation plotting frameworks to DNA Features Viewer. As a benchmark we use a GFF annotations file featuring 3 gene expression units, as shown in Table SI1, and we will show how each framework plots the record with minimal configuration.

chrom1	custom	backbone	0	4400	.	+	.	Name=backbone
chrom1	custom	promoter	10	58	.	+	.	Name=P1
chrom1	custom	gene	67	948	.	+	.	Name=geneA
chrom1	custom	terminator	949	1000	.	+	.	Name=T1
chrom1	custom	promoter	1124	1125	.	+	.	Name=P2
chrom1	custom	gene	1134	4300	.	+	.	Name=another gene with an extremely very long name
chrom1	custom	terminator	4301	4350	.	+	.	Name=T2
chrom1	custom	promoter	4500	4650	.	+	.	Name=P3
chrom1	custom	gene	4651	6300	.	+	.	Name=GFP
chrom1	custom	terminator	6301	6450	.	+	.	Name=T3

Table SI1: Annotations in the `plasmid.gff` file used as a benchmark in this section (the actual file contains exactly this information, with one entry per line and tabulations separating each entry's columns).

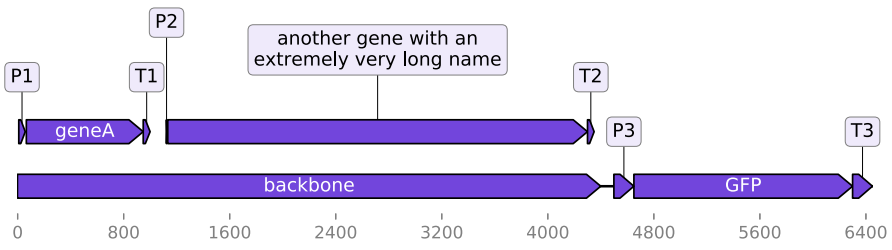
A1. Plotting with DNA Features Viewer

We first plot the record using DNA Feature Viewer, without any configuration or customization:

Code:

```
from dna_features_viewer import BiopythonTranslator
ax = BiopythonTranslator.quick_class_plot("plasmid.gff", figure_width=9)
ax.figure.savefig('dfv.svg', bbox_inches='tight')  # SAVE AS SVG
```

Result:



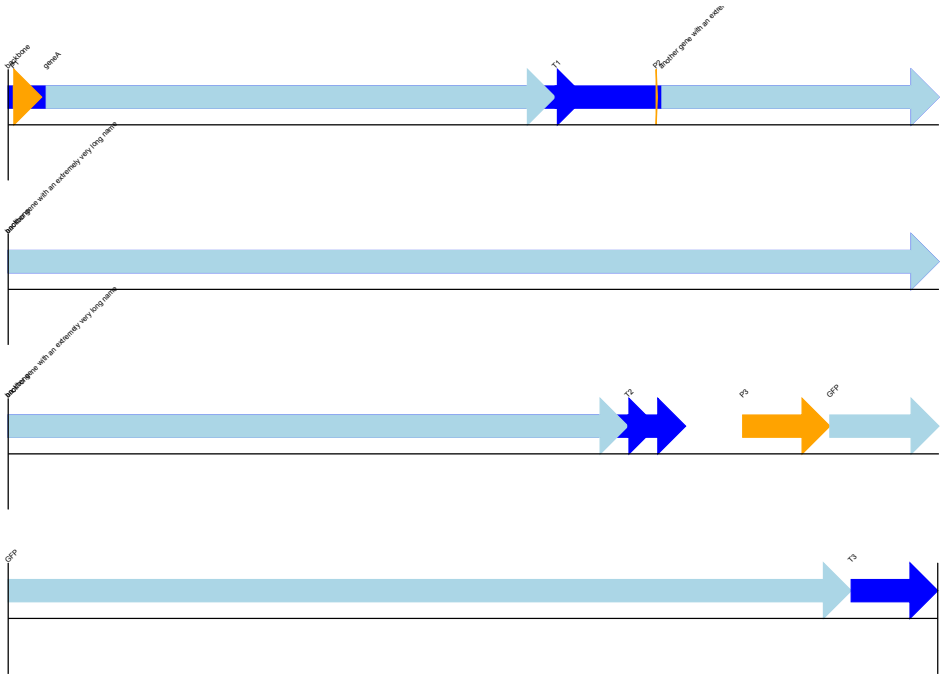
A2. Plotting with the Biopython plotting module

The script below is a variant from a script proposed in the official Biopython Cookbook tutorial (**http://biopython.org/DIST/docs/tutorial/Tutorial.html**):

Code:

```
from reportlab.lib import colors
from Bio.Graphics import GenomeDiagram
from Bio import SeqIO
from dna_features_viewer import load_record

record = load_record("plasmid.gff")
gd_diagram = GenomeDiagram.Diagram()
gd_track_for_features = gd_diagram.new_track(1, name="features")
gd_feature_set = gd_track_for_features.new_set()
colors = [colors.blue, colors.orange, colors.lightblue]

for feature in record.features:
    color = colors[len(gd_feature_set) % 3]
    gd_feature_set.add_feature(feature, color=color, label=True, sigil="ARROW")
gd_diagram.draw(format="linear", orientation="landscape", pagesize='A4',
                fragments=4, start=0, end=len(record))
gd_diagram.write("biopython.svg", "SVG")
```

Result:



Biopython's plotting module requires the user to specify colors for each feature separately (here we manually alternate between 3 colors, following the example of the Biopython tutorial, so that successive features can be distinguished). All features are plotted on a single line (unless the user places them manually on different tracks or different figures), causing overlapping features to collide. Labels are small , making the figure hard to read, although this also decreases the chances of label collisions (text collisions can still be seen at the beginning of lines 2 and 3).

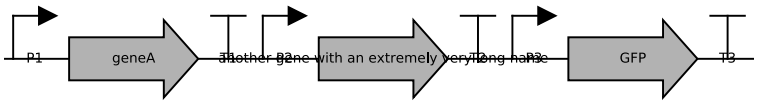
A3. Plotting with DnaPlotLib

Here we use DnaPlotLib's builtin `load_design_from_gff` method to plot the GFF file's annotations with DnaPlotLib:

Code:

```
import dnaplotlib as dpl
import matplotlib.pyplot as plt
from matplotlib import gridspec

design = dpl.load_design_from_gff("plasmid.gff", "chrom1", region=[0, 6451])

# Create the DNAplotlib renderer
dr = dpl.DNARenderer()
part_renderers = dr.SBOL_part_renderers()

# Create the figure
fig, ax_dna = plt.subplots(1, figsize=(10.0, 1.2))

# Redender the DNA to axis
start, end = dr.renderDNA(ax_dna, design, part_renderers)
ax_dna.set_xlim([start, end])
ax_dna.set_ylim([-15, 15])
ax_dna.set_aspect("equal")
ax_dna.axis("off")
ax_dna.figure.savefig("with_dnaplotlib.svg", bbox_inches='tight')
```

Result:



The DnaPlotLib library focuses on the display of the funtional genetic elements of a sequence (promoters, coding sequences, terminators, etc), and allows a high level of manual customization to produce publication-quality plots. However, we see in this example that it less adapted to the general display of annotations from arbitrary GFF or genbank records. The *"backbone"* annotation overlapping

with the first two expression units is not displayed (information is lost) and collisions between text and other elements are not automatically avoided (some examples in the DnaPlotLib documentation show that it is possible to manually provide user-selected offsets in the Python script to place features and texts on different levels to avoid collisions, but this is not automated).

A4. The Biograpy library

The BiograPy library (A. Pierleoni, unpublished, **http://apierleoni.github.io/BioGraPy/tutorial.html**) and its most recent fork (M.O. Weber, unpublished, **https://github.com/webermarcolivier/BioGraPy**), allow users to define features which are then automatically placed in a plot so as to avoids collisions between overlapping features.

Unfortunately the libraries seem to rely on outdated dependencies (the latest code contributions to the projects are from 2016 and 2017) and we did not manage to run them on our example record. Therefore we are only showing screenshots from the projects' websites in Figure SI1 below.

Among the notable differences with DNA Features Viewer, the labels are always placed inside or right under their corresponding feature's arrow, which can be problematic for sequences with a high density of small annotations. The library does not feature any equivalent of DNA Features Viewer's `BiopythonTranslator` to automatically convert genbank records to graphic records.

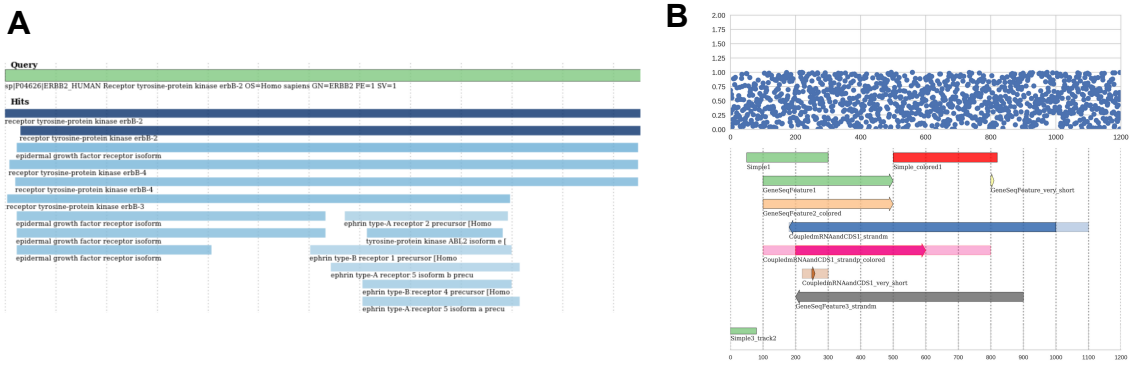
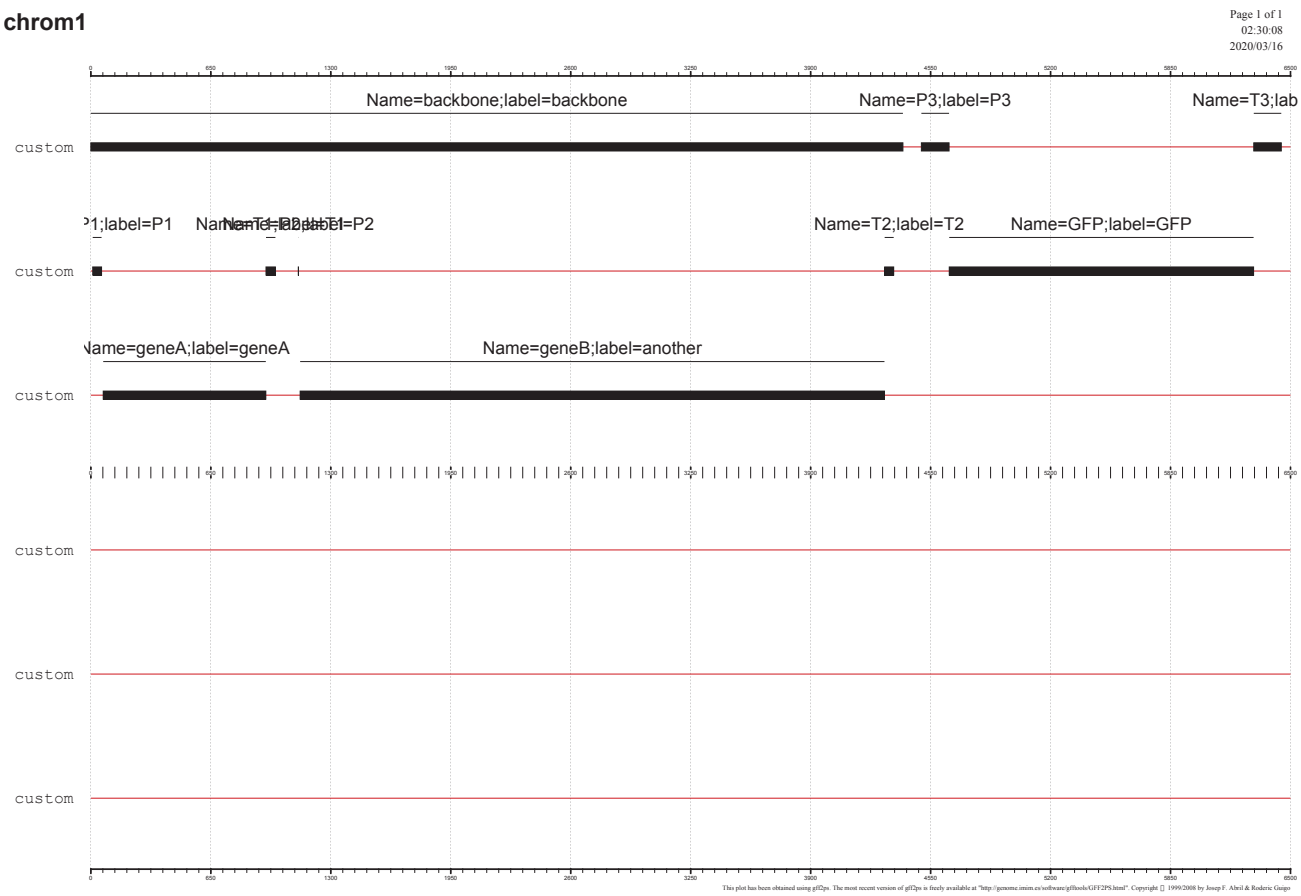


Figure SI1: Sample outputs of the Biograpy library, generated by the original project (panel A) and its more recent fork (panel B). See the links provided in this paragraph for the source of these figures.

A5. The gff2ps command-line interface

The figure below was generated with the command `gff2ps plasmid.gff`. While `gff2ps` automatically places overlapping features on different tracks, it does not avoid label collisions (as can be seen on track 2).



Summary table

Framework	Input	Customization	Auto. Collision avoidance	Output
DNA Features Viewer	Biopython record (can be generated from GFF. Genbank, ...)	colors, fonts, line widths	features arrows and features labels	PDF, SVG, PNG…
DnaPlotLib	GFF	colors, fonts, line widths, shapes	none	PDF, SVG, PNG…
Biopython	Biopython record (can be generated from GFF. Genbank, ...)	colors, shapes	features arrows	PDF, SVG, PNG…
Biograpy	Biopython record (can be generated from GFF. Genbank, ...)	colors, shapes	features arrows	PDF, SVG, PNG…
gff2ps	GFF	colors, shapes	features arrows	PS

B. Features and annotations positioning algorithm

Problem definition

This section describes how sequence features arrows and labels are vertically positioned by DNA Feature Viewer to avoid any collision (i.e. the superimposition of two graphical elements).

Every graphical element has a set horizontal coordinates (x_1, x_2). For feature arrows, these coordinates correspond to the feature's start- and end-position in the sequence. For labels, it corresponds to the horizontal coordinates of the text after centering on the middle of the feature's location. An element is said to be horizontally overlapping with another element at position (x'_1, x'_2) if the two segments overlap, which is equivalent to:

$$(\text{Overlapping Condition}) \quad x_1 < x'_1 < x_2 \quad \text{or} \quad x'_1 < x_1 < x'_2$$

As of DNA Features Viewer v2.3, each element is placed vertically a on certain level $v > 0$, each level having approximately the same height as a line of text. Thus, two horizontally overlapping features placed will collide if and only if they are also placed on the same level.

The placement problem consists, for given a set of graphic elements, in determining a level v for each element, so that (1) no horizontally overlapping elements have the same level, and (2) the largest level v among all elements is as small as possible (to keep the plot compact).

Formulation as a graph coloring problem

The DNA Feature Viewer algorithm first builds a graph where each node represents a graphical element, and an edge between two nodes indicate that the corresponding elements are horizontally overlapping. The problem is now to find a *coloring* of each node with a level v that differs from the levels of all neighbors in the graph, to avoid collisions. This is a classical graph colorouring problem, which is known to be NP-complete (Brelax 1979), i.e. computationally intensive for large problems. Therefore, the algorithm uses *greedy coloring*, where it sequentially attributes the lowest available level to each element (starting with the widest elements so the larger features appear at the bottom of the plot):

- **For each** element:
    - List all the element's neighbors in the graph for which a level has been set.
    - Set the element's level to 0
    - **While** the element collides with any neighbor in the graph:
        - Increase the element's level by one.

Many small improvements are done to improve graph readability. First, larger features are considered before smaller ones in the iteration loop. As a result, the larger features always appear at the bottom of the plot. Second, all features arrows are attributed a level before all feature labels, so that the labels

"float" on top of the feature arrows. Third, multi-line labels are taken into account, as explained in the next section.

Support for multi-line labels

Labels may have a certain number N of lines (feature arrows can be considered has being on a single line, i.e. $N = 1$). In this context, two horizontally overlapping elements, with respectively N lines at level v, and N' lines at v', will collide if their levels are not sufficiently spread appart, or more precisely when the following condition (also illustrated in Figure SI2) is met:

$$(\text{Collision Condition}) \quad |v - v'| < \frac{N + N'}{2}$$

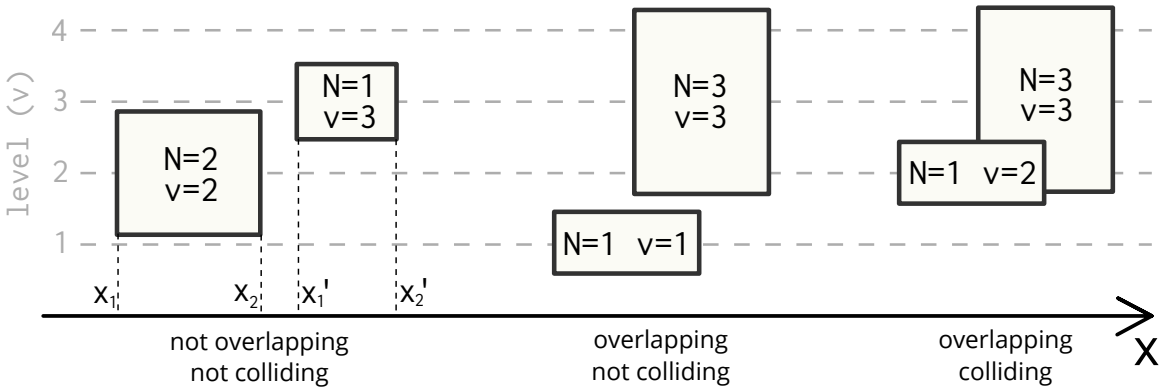


Figure SI2: Graphical elements with varying level (v) and number of lines (N) as defined in the main text.

C. Python code snippets for Figure 1

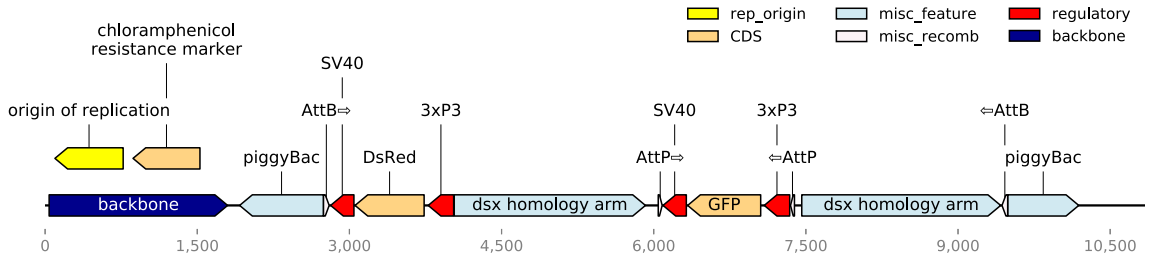
This sections provides the Python code for the plots shown in Figure 1 of the main text. Note that these snippets can also in the *Examples* section of the online project (**https://github.com/Edinburgh-Genome-Foundry/DnaFeaturesViewer/tree/master/examples**).

While the scripts to generate the different panels can be run in any order, they all require to first install the DNA Features Viewer library version 3.0 or more recent (either using the `pip` installation system via `pip install dna_features_viewer`, or by downloading and installing the library locally, as explained in the project documentation). It is also necessery to download (from NCBI.org) the plasmid record that will be used as a sample, using the code below:

```
from Bio import Entrez, SeqIO
from dna_features_viewer import annotate_biopython_record

Entrez.email = "dna_features_viewer@example.com"
handle = Entrez.efetch(
    db="nucleotide", id=1473096477, rettype="gb", retmode="text"
)
record = SeqIO.read(handle, "genbank")
annotate_biopython_record(
    record, location=(40, 1800), feature_type="backbone", label="backbone"
)
record.features = [
    f for f in record.features if f.type not in ["gene", "source"]
]
SeqIO.write(record, "plasmid.gb", "genbank")
```

Panel A (linear view)

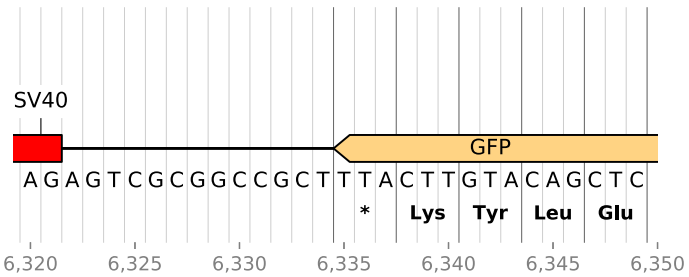
The script below defines a `CustomTranslator` class setting annotations colors based on feature type. All floating feature labels are written on a white background with no text box line.



```python
from dna_features_viewer import BiopythonTranslator


class CustomTranslator(BiopythonTranslator):

    # Label fields indicates the order in which annotations fields are
    # considered to determine the feature's label
    label_fields = ["label", "note", "name", "gene"]

    def compute_feature_legend_text(self, feature):
        return feature.type

    def compute_feature_color(self, feature):
        return {
            "rep_origin": "yellow",
            "CDS": "#ffd383",  # light orange
            "regulatory": "red",
            "misc_recomb": "#fbf3f6",  # pink
            "misc_feature": "#d1e9f1",  # light blue
            "backbone": "darkblue",
        }[feature.type]

    def compute_feature_box_color(self, feature):
        return "white"

    def compute_feature_box_linewidth(self, feature):
        return 0


translator = CustomTranslator()
graphic_record = translator.translate_record("plasmid.gb")
ax, _ = graphic_record.plot(figure_width=13, strand_in_label_threshold=7)
graphic_record.plot_legend(ax=ax, loc=1, ncol=3, frameon=False)
ax.figure.savefig("A_linear_plot.svg", bbox_inches="tight")
```

Panel B (detail view)

The script below imports the `CustomTranslator` defined in the previous section and uses it to display a cropped segment of the record (between indices 6320 and 6350), this time with nucleotide and amino-acid sequences overlaid on the figure.

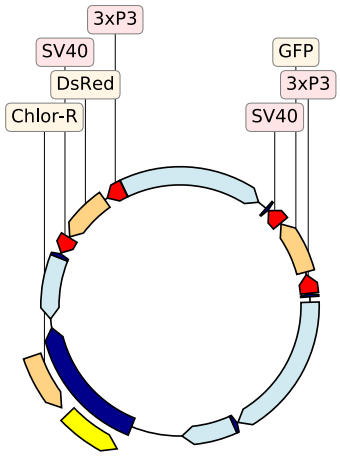


```
from A_linear_plot import CustomTranslator

translator = CustomTranslator()
graphic_record = translator.translate_record("plasmid.gb")
cropped_record = graphic_record.crop((6320, 6350))
cropped_record.ticks_resolution = 5  # One tick every 5 nucleotides
ax, _ = cropped_record.plot(figure_width=6)
cropped_record.plot_sequence(ax, guides_intensity=0.2)
cropped_record.plot_translation(
    ax=ax, location=(6335, 6350, -1), fontdict={"weight": "bold"}
)
ax.figure.savefig("B_detail_plot.svg", bbox_inches="tight")
```

Panel C (circular view)

The script below defines an `ExpressionUnitTranslator` class which only labels coding sequences and regulatory elements. In addition, the script uses the `CircularGraphicRecord` class to plot the record, resulting in a circular plot.

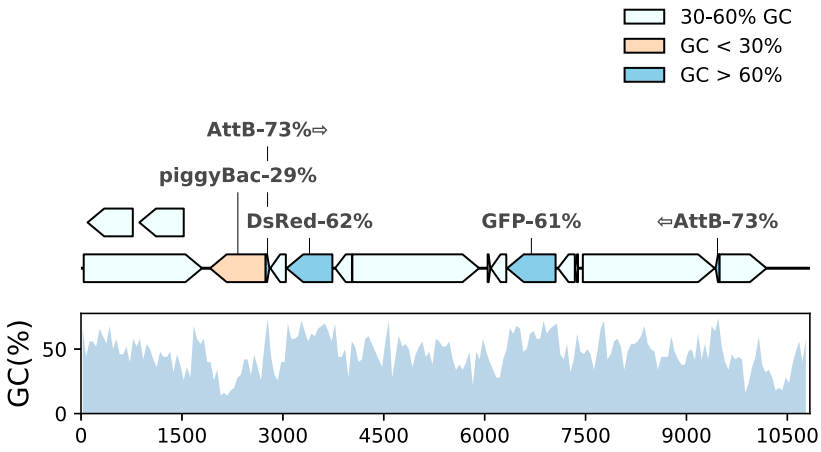
```
from dna_features_viewer import BiopythonTranslator, CircularGraphicRecord

class ExpressionUnitTranslator(BiopythonTranslator):

    def compute_feature_color(self, feature):
        color_map = {
            "rep_origin": "yellow",
            "CDS": "#ffd383",  # light orange
            "regulatory": "red",
            "misc_recomb": "darkblue",
            "misc_feature": "#d1e9f1",  # light blue
            "backbone": "darkblue",
        }
        return color_map[feature.type]

    def compute_feature_label(self, feature):
        if feature.type not in ["CDS", "regulatory"]:
            return None
        else:
            return BiopythonTranslator.compute_feature_label(self, feature)


translator = ExpressionUnitTranslator()
graphic_record = translator.translate_record(
    "plasmid.gb", record_class=CircularGraphicRecord
)
graphic_record.top_position = 4800  # sequence index appearing at the top
ax, _ = graphic_record.plot(figure_width=4)
ax.figure.savefig("C_circular_display.svg", bbox_inches="tight")
```



Panel D (GC% view)

The script below plots the sequence record alongside a profile of the local GC content. We first define a custom `GCIndicatingTranslator` class which separates features in regions with over 60% or under 30% GC, then we use Matplotlib to create two vertically-aligned subplots, for the record plot and GC content plot, respectively.

Note that `GCIndicatingTranslator` expects the features of the genbank record to be translated to have a `gc%` attribute. In the script, this attribute is computed for all features prior to creating the `GCIndicatingTranslator` instance.



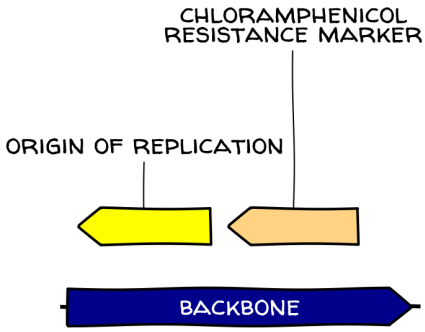

```
from dna_features_viewer import BlackBoxlessLabelTranslator, load_record
import matplotlib.pyplot as plt
import numpy as np
```

```
class GCIndicatingTranslator(BlackBoxlessLabelTranslator):
    def compute_feature_legend_text(self, feature):
        if feature.qualifiers["gc%"] < 30:
            return "GC < 30%"
        elif feature.qualifiers["gc%"] < 60:
            return "30-60% GC"
        else:
            return "GC > 60%"

    def compute_feature_color(self, feature):
        return {
            "GC < 30%": "peachpuff",
            "30-60% GC": "azure",
            "GC > 60%": "skyblue",
        }[self.compute_feature_legend_text(feature)]

    def compute_feature_fontdict(self, feature):
        return dict(size=10, weight="bold", color="#494949")

    def compute_feature_label(self, feature):
        if not (30 < feature.qualifiers["gc%"] < 60):
            normal_label = super().compute_feature_label(feature)
            return normal_label + "-%d%%" % feature.qualifiers["gc%"]


def gc_content(sequence):
    return 100.0 * len([c for c in sequence if c in "GC"]) / len(sequence)


# DISPLAY THE SEQUENCE MAP

fig, (ax1, ax2) = plt.subplots(
    2, 1, figsize=(6, 3.5), sharex=True, gridspec_kw={"height_ratios": [3, 1]},
)
record = load_record("plasmid.gb")
for feature in record.features:
    feature.qualifiers["gc%"] = gc_content(feature.location.extract(record))
translator = GCIndicatingTranslator()
graphic_record = translator.translate_record(record)

graphic_record.plot(ax=ax1, with_ruler=False, strand_in_label_threshold=7)
graphic_record.plot_legend(ax=ax1, loc=1, frameon=False)

# DISPLAY THE GC% PROFILE ALONG THE SEQUENCE

window_size = 50
windowed_gc_content = [
    gc_content(record.seq[i : i + window_size])
    for i in range(len(record.seq) - window_size)
]
indices = np.arange(len(record.seq) - window_size) + 25

ax2.fill_between(indices[::50], windowed_gc_content[::50], alpha=0.3)
ax2.set_ylabel("GC(%)", fontsize=14)
ax2.set_ylim(bottom=0)
```

```
# SAVE THE FIGURE

fig.tight_layout()
fig.savefig("D_display_with_gc_content.svg", bbox_inches="tight")
```

Panel E (sketch effect)

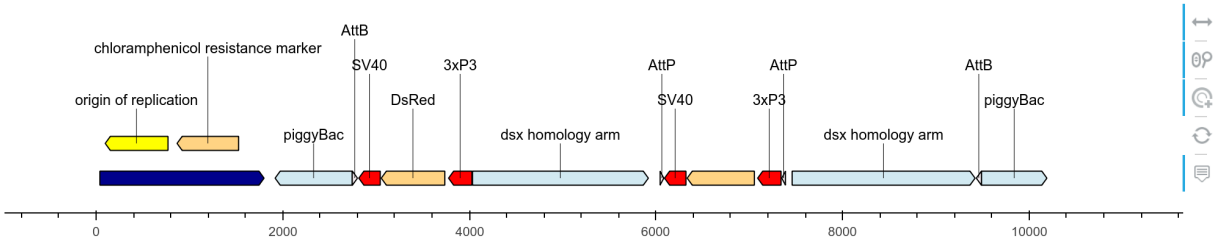
In this script we plot a cropped segment of the record, using a custom font, and the `path.sketch` filter of Matplotlib to introduce randomness in the line drawing, creating a *hand-drawing* effect.




```
from matplotlib import rc_context
from A_linear_plot import CustomTranslator

rc_context({"path.sketch": (1.5, 300, 1)})  # scale, length, randomness


class CustomTranslatorVariant(CustomTranslator):
    def compute_feature_fontdict(self, feature):
        return {"family": "Walter Turncoat"}


translator = CustomTranslatorVariant()
graphic_record = translator.translate_record("plasmid.gb")
cropped_record = graphic_record.crop((0, 1850))
ax, _ = cropped_record.plot(figure_width=2.5, with_ruler=False)
ax.figure.savefig("E_cartoon_plot.png", dpi=300, bbox_inches="tight")
```

Panel F (interfactive plot)

In this script we create an HTML page featuring an interactive plot of the record using the Bokeh framework.




```
from bokeh.embed import file_html
from bokeh.resources import CDN
from A_linear_plot import CustomTranslator

translator = CustomTranslator()
graphic_record = translator.translate_record("plasmid.gb")
bokeh_plot = graphic_record.plot_with_bokeh(figure_width=10, figure_height=2)
html = file_html(bokeh_plot, CDN, "my plot")
with open("F_bokeh_plot.html", "w") as f:
    f.write(html)
```

D. Multiline, multi-page plot

The script below plots the full plasmid of the main text into a multi-line, multi-pages PDF file. The final file features 14 pages with 10 lines of 80 nucleotides per page (the three first pages are show below).

```
from A_linear_plot import CustomTranslator

translator = CustomTranslator()
graphic_record = translator.translate_record("./plasmid.gb")
graphic_record.plot_on_multiple_pages(
    "multiline_plot.pdf",
    nucl_per_line=80,
    lines_per_page=10,
    plot_sequence=True
)
```

Bibliography

- Brélaz, D. (1979). New Methods to Color the Vertices of a Graph. Communications of the ACM. https://doi.org/10.1145/359094.359101